Towards Practical Visual Search Engine Within Elasticsearch

Cun (Matthew) Mu
Jet.com/Walmart Labs
Hoboken, NJ
matthew.mu@jet.com

Jun (Raymond) Zhao
Jet.com/Walmart Labs
Hoboken, NJ
raymond@jet.com

Guang Yang
Jet.com/Walmart Labs
Hoboken, NJ
guang@jet.com

Jing Zhang
Jet.com/Walmart Labs
Hoboken, NJ
jing@jet.com

Zheng (John) Yan
Jet.com/Walmart Labs
Hoboken, NJ
john@jet.com

ABSTRACT

In this paper, we describe our end-to-end content-based image retrieval system built upon Elasticsearch, a well-known and popular textual search engine. As far as we know, this is the first time such a system has been implemented in eCommerce, and our efforts have turned out to be highly worthwhile. We end up with a novel and exciting visual search solution that is extremely easy to be deployed, distributed, scaled and monitored in a cost-friendly manner. Moreover, our platform is intrinsically flexible in supporting multimodal searches, where visual and textual information can be jointly leveraged in retrieval.

The core idea is to encode image feature vectors into a collection of string tokens in a way such that closer vectors will share more string tokens in common. By doing that, we can utilize Elasticsearch to efficiently retrieve similar images based on similarities within encoded sting tokens. As part of the development, we propose a novel vector to string encoding method, which is shown to substantially outperform the previous ones in terms of both precision and latency.

First-hand experiences in implementing this Elasticsearch-based platform are extensively addressed, which should be valuable to practitioners also interested in building visual search engine on top of Elasticsearch.

CCS CONCEPTS

• **Information systems → Image search**; • **Applied computing → Online shopping**;

KEYWORDS

Elasticsearch, visual search, content-based image retrieval, multimodal search, eCommerce

ACM Reference Format:
Cun (Matthew) Mu, Jun (Raymond) Zhao, Guang Yang, Jing Zhang, and Zheng (John) Yan. 2018. Towards Practical Visual Search Engine Within Elasticsearch. In *Proceedings of ACM SIGIR Workshop on eCommerce (SIGIR 2018 eCom).* ACM, New York, NY, USA, 8 pages.

SIGIR 2018 eCom, July 2018, Ann Arbor, Michigan, USA

1 INTRODUCTION

Elasticsearch [22], built on top of Apache Lucene library [4, 15, 38], is an *open-source*, *real-time*, *distributed* and *multi-tenant* textual search engine. Since its first release in February 2010, Elasticsearch has been widely adopted by eCommerce websites (e.g., Ebay, Etsy, Jet, Netflix, Grubhub) to successfully help customers discover products based on the textual queries they requested [13, 57] .

But *a picture is more than often worth a thousand words.* With the explosive usage of phone cameras, *content-based image retrieval* [16] is increasingly demanded from customers. Especially for categories like furniture, fashion and lifestyle (where buying decisions are largely influenced by products' visual appealingness), uploading a picture of the product they like could be substantially more specific, expressive and straightforward than elaborating it into abstract textual description.

Finding images relevant with the uploaded picture tends to be much more involved and vaguer than retrieving documents matching keywords [45, 48, 58] typed into the search box, as words (by themselves) are substantially more semantic and meaningful than image pixel values. Fortunately, modern AI techniques, especially the ones developed in the field of deep learning [3, 21], have made incredible strides in image feature extraction [17, 32, 39, 42–44, 59, 60] to embed images as points in *high-dimensional* Euclidean space, where similar images are located nearby. So, given a query image, we can simply retrieve its visually similar images by *finding its nearest neighbors in this high-dimensional feature space.* However, Elasticsearch, as an *inverted-index-based search engine*, is not much empowered to accomplish this mathematically straightforward operation in an efficient manner (though efforts [6, 7, 19, 30, 36] have been made successfully in finding nearest neighbors over spaces of much lower dimension), which significantly limits the applicability of its nicely designed engineering system as well as the huge volume of product metadata already indexed into its database (for textual search). The gist of the paper is to conquer this difficulty, and thus make it feasible to conduct visual search within Elasticsearch.

In this paper, we describe our end-to-end visual search platform built upon Elasticsearch. As far as we know, this

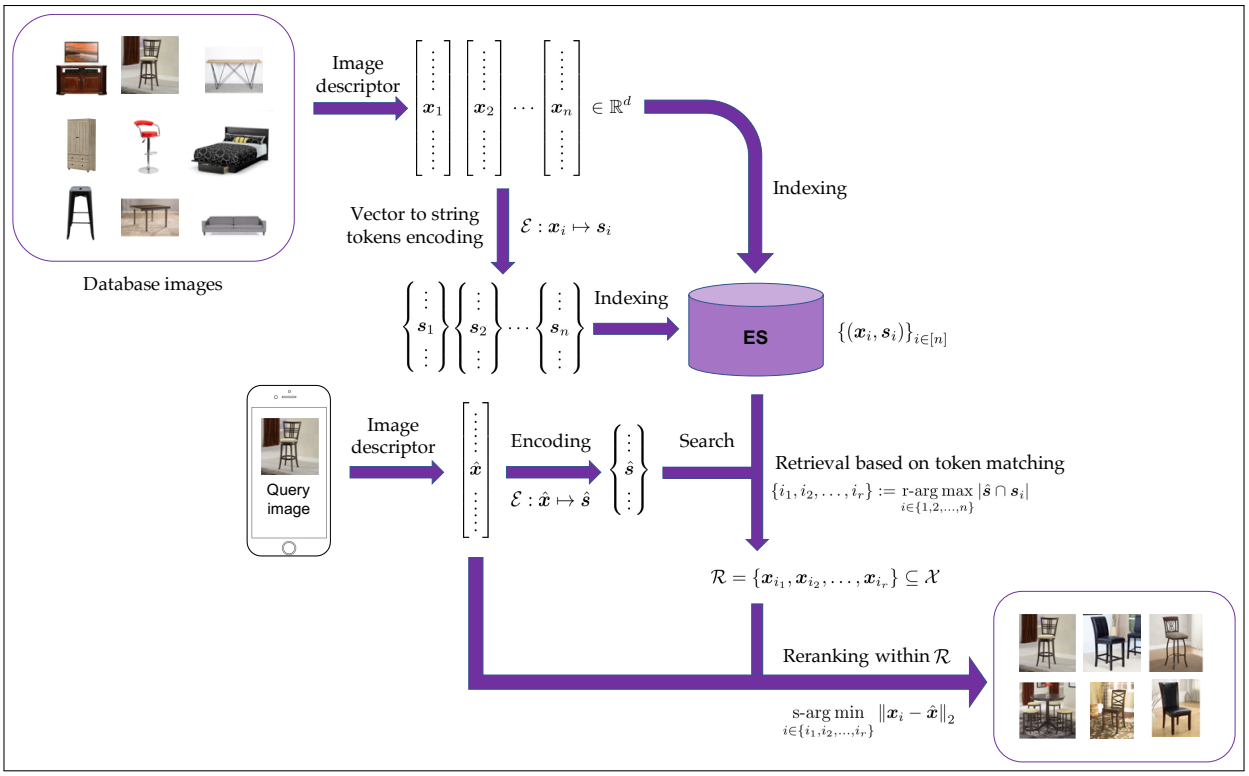

Figure 1: Pipeline of our visual search system within Elasticsearch. The image vectors and their encoded string tokens are indexed together into Elasticsearch. At search time, the query vector $\hat{\boldsymbol{x}}$ will be first encoded into string tokens $\hat{\boldsymbol{s}}$, based on which a small candidate set $\mathcal{R}$ is retrieved. We will then re-rank vectors in $\mathcal{R}$ according to their exact Euclidean distances with $\hat{\boldsymbol{x}}$, and output the top ones as our final visual search outcome.

is *the first attempt* to achieve this goal and our efforts turn out to be quite worthwhile. By taking advantage of the mature engineering design from Elasticsearch, we end up with a visual search solution that is extremely easy to be *deployed*, *distributed*, *scaled* and *monitored*. Moreover, due to Elasticsearch's disk-based (and partially memory cached) inverted index mechanism, our system is quite *cost-effective*. In contrast to many existing systems (using *hashing-based* [2, 20, 23, 33, 54–56] or *quantization-based* [18, 25–27, 29] *approximate nearest neighbor* (ANN) methods), we do not need to load those millions of (high-dimensional and dense) image feature vectors into RAM, one of the most expensive resources in large-scale computations. Furthermore, by integrating textual search and visual search into one engine, both types of product information can now be shared and utilized seamlessly in a single index. This paves a coherent way to support *multimodal searches*, allowing customers to express their interests in a variety of textual requests (e.g., keywords, brands, attributes, price ranges) jointly with visual queries, at which most of existing visual search systems fall short (if not impossible).

Since the image preprocessing step and the image feature extraction step involved in our system are standard and independent of Elasticsearch, in this paper we address more towards how we empower Elasticsearch to retrieve close image feature vectors, i.e., *the Elasticsearch-related part of the visual system*. Our nearest neighbor retrieval approach falls under the general framework recently proposed by Rygl et al. [47]. The core idea is to create *text documents* from *image feature vectors* by encoding each vector into a collection of string tokens in a way such that *closer vectors will share more string tokens in common*. This enables Elasticsearch to approximately retrieve neighbors in image feature space based on their encoded textual similarities. The quality of the encoding procedure (as expected) is extremely critical to the success of this approach. In the paper, we propose a noval scheme called *subvector-wise clustering encoder*, which substantially outperforms the element-wise rounding one proposed and examined by Rygl et al. [47] and Ruzicka et al. [46], in terms of both *precision* and *latency*. Note that our methodology should be generally applicable to any full-text search engine (e.g., Solr [51], Sphinx [1]) besides Elasticsearch, but in the paper we do share a number of Elasticsearch-specific implementation tips based on our first-hand experience, which should be valuable to practitioners interested in building their own visual search system on top of Elasticsearch.

The rest of the paper is organized as follows. In Section 2, we describe the general pipeline of our visual search system, and highlight a number of engineering tweaks we found useful when implementing the system on Elasticsearch. In Section 3 and 4, we focus on how to encode an image feature vector into

a collection of string tokens—the most crucial part in setting up the system. In Section 3, we first review the element-wise rounding encoder and address its drawbacks. As a remedy, we propose a new encoding scheme called subvector-wise clustering encoder, which is empirically shown in Section 4 to much outperform the element-wise rounding one.

2 GENERAL FRAMEWORK OF VISUAL SEARCH WITHIN ELASTICSEARCH

The whole pipeline of our visual search engine is depicted in Figure 1, which primarily consists of two phases: indexing and searching.

Indexing. Given image feature vectors

$$\mathcal{X} := \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\} \subseteq \mathbb{R}^d, \tag{2.1}$$

we will first encode them into string tokens

$$\mathcal{S} := \{\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_n\}, \tag{2.2}$$

where $\boldsymbol{s}_i := \mathcal{E}(\boldsymbol{x}_i)$ for some encoder $\mathcal{E}(\cdot)$ converting a d-dimensional vector into a collection of string tokens of cardinality m. The original numerical vectors $\mathcal{X}$ and encoded tokens $\mathcal{S}$, together with their textual metadata (e.g, product titles, prices, attributes), will be all indexed into the Elasticsearch database, to wait for being searched.

Searching. Conceptually, the search phase consists of two steps: *retrieval* and *reranking*. Given a query vector $\hat{\boldsymbol{x}}$, we will first encode it into $\hat{\boldsymbol{s}} := \mathcal{E}(\hat{\boldsymbol{x}})$ via the same encoder used in indexing, and retrieve r ($r \ll n$) most similar vectors $\mathcal{R} := \left\{\boldsymbol{x}_{i_1}, \boldsymbol{x}_{i_2}, \ldots, \boldsymbol{x}_{i_r}\right\}$ as candidates based on the overlap between the string token set $\hat{\boldsymbol{s}}$ and the ones in $\{\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_n\}$, i.e.,

$$\{i_1, i_2, \ldots, i_r\} = \underset{i \in \{1,2,\ldots,n\}}{\text{r-arg max}} \ |\hat{\boldsymbol{s}} \cap \boldsymbol{s}_i|. \tag{2.3}$$

We will then re-rank vectors in the candidate set $\mathcal{R}$ according to their *exact* Euclidean distances with respect to the query vector $\hat{\boldsymbol{x}}$, and choose the top-s ($s \leq r$) ones as the final visual search result to output, i.e.,

$$\underset{i \in \{i_1, i_2, \ldots, i_r\}}{\text{s-arg min}} \ \|\boldsymbol{x}_i - \hat{\boldsymbol{x}}\|_2 \,. \tag{2.4}$$

As expected, the choice of $\mathcal{E}(\cdot)$ is extremely critical to the success of the above approach. A good encoder $\mathcal{E}(\cdot)$ should encourage image feature vectors closer in Euclidean distance to share more string tokens in common, so that the retrieval set $\mathcal{R}$ obtained from the optimization problem (2.3) could contain enough meaning candidates to be fed into the exact search in (2.4). We will elaborate and compare different choices of encoders in the next two sections (Section 3&4).

Implementation. In this part, we will address how we implement the retrieval and reranking steps in the searching phase efficiently within just one JSON-encoded request body (i.e., JSON 1), which instructs the Elasticsearch server to compute (2.3) and (2.4) and then return the visual search result in a desired order (via Elasticsearch's RESTful API over HTTP).

For the retrieval piece, we construct a *function score query* [9] to rank database images based on (2.3). Specifically, our function score query (lines 3-29 in JSON 1) consists of m score functions, each of which is a *term filter* [14] (e.g., lines 6-14 in JSON 1) to check whether the encoded feature token $\hat{s}_i$ from the query image is being matched or not. With all the m scores being summed up (line 26 in JSON 1) using the same weight (e.g., lines 13 and 23 in JSON 1), the ranking score for the database images are calculated exactly as the number of feature tokens they overlap with the ones in $\hat{\boldsymbol{s}}$.

For the reranking piece, our initial trial is to fetch the top-r image vectors from the retrieval step, and calculate (2.4) to re-rank them outside Elasticsearch. But this approach prevents our visual system from being an end-to-end one within Elasticsearch, and thus makes it hard to leverage many useful microservices (e.g., pagination) provided by Elasticsearch. More severely, this vanilla approach introduces substantial latency in communication as thousands of high-dimensional and dense image embedding vectors have to be transported out of Elasticsearch database. As a remedy, we design a *query rescorer* [12] (lines 30-52 in JSON 1) within Elasticsearch to execute a second query on the top-r database image vectors returned from the function score query, to tweak their scores and re-rank them based on their exact Euclidean distances with the query image vector. In specific, we implement a custom *Elasticsearch plugin* [10] (lines 35-47 in JSON 1) to compute the *negation* of the Euclidean distance between query image vector and the one from database. As Elasticsearch will rank the result based on the ranking score from high to low, the output will be in the desired order from the smallest distance to the largest one.

Multimodal search. More often than not, scenarios more complicated than visual search will be encountered. For instance, a customer might be fascinated with the design and style of an armoire at her friend's house, but she might want to change its color to be better aligned with her own home design or want the price to be within her budget (see Figure 2). Searching using the picture snapped is most likely in vain. To better enhance customers' shopping experiences, a visual search engine should be capable of retrieving results as *a joint outcome* by taking both the visual and textual requests from customers into consideration. Fortunately, our Elasticsearch-based visual system can immediately achieve this with one or two lines modifications in JSON 1. In particular, filters can be inserted within the function score query to search only among products of customers' interests (e.g., within certain price range [11], attributes, colors). Moreover, general full-text query [8] can also be handled, score of which can be blended with the one from visual search in a weighted manner.

3 VECTOR TO STRING ENCODING

The success of our approach hinges upon the quality of the encoder $\mathcal{E}(\cdot)$, which ideally should encourage closer vectors to share more sting tokens in common, so that the retrieval set $\mathcal{R}$ found based on token matching contains enough meaningful candidates. In the following, we first review the element-wise rounding encoder proposed by Rygl et al. [47], and discuss its potential drawbacks. As a remedy, we propose a novel encoding scheme called subvector-wise clustering encoder.

JSON 1 Request body for visual search in Elasticsearch 6.1

```
{
  "size": s,
  "query": {
    "function_score": {
      "functions": [
      {
        "filter": {
          "term": {
            "image_encoded_tokens":
              "query_encoded_token_1"
          }
        },
        "weight": 1
      },
      ...,
      {
        "filter": {
          "term": {
            "image_encoded_tokens":
              "query_encoded_token_m"
          }
        },
        "weight": 1
      }
      ],
      "score_mode": "sum",
      "boost_mode": "replace"
    }
  },
  "rescore": {
    "window_size": r,
    "query": {
      "rescore_query": {
        "function_score": {
        "script_score": {
          "script": {
            "lang": "custom_scripts",
            "source": "negative_euclidean_distance",
            "params": {
              "vector_field": "image_actual_vector",
              "query_vector":
                [0.1234, -0.2394, 0.0657, ...]
            }
          }
        },
        "boost_mode": "replace"
      }
    },
    "query_weight": 0,
    "rescore_query_weight": 1
    }
  }
}
```

Figure 2: Illustration of multimodal search. Armoire is searched using image query jointly with color/price range specified by the customer. Our Elasticsearch-based visual search engine can be easily tailored to handle complicated business requests like the above by adding filters (e.g., *term filter* [14], *range filter* [11]) to JSON 1.

3.1 Element-wise Rounding

Proposed and examined by Rygl et al. [47] and Ruzicka et al. [46], the element-wise rounding encoder rounds each value in the numerical vector to p decimal places (where $p \geq 0$ is a fixed integer), and then concatenates its positional information and rounded value as the string tokens.

Example 1. *For a vector* $\boldsymbol{x} = [0.1234, -0.2394, 0.0657]$, *rounding to two decimal places (i.e.,* $p = 2$*) produces string tokens of* $\boldsymbol{x}$ *as*

$$\boldsymbol{s} = \{\text{“pos1val0.12”}, \text{“pos2val-0.24”}, \text{“pos3val0.07”}\}\,.$$

The encoded positional information is essential for the inverted-index-based search system to match (rounded) values at the same position without confusion. Suppose on the other hand, positional information is ignored, and thus

$$\boldsymbol{s} = \{\text{“val0.12”}, \text{“val-0.24”}, \text{“val0.07”}\}\,.$$

Then the attribute “val0.12” could be mistakenly matched by another encoded token even when it is not produced from the first entry.

For a high-dimensional vector $\boldsymbol{x} \in \mathbb{R}^d$, this vanilla version of the element-wise rounding encoder will generate a large collection of string tokens (essentially with $|\mathcal{E}(\boldsymbol{x})| = d$), which makes it infeasible for Elasticsearch to compute (2.3) in real time.

Filtering. As a remedy, Rygl et al. [47] presents a useful filtering technique to sparsify the string tokens. In specific, only top-m entries in terms of magnitude are selected to create rounding tokens.

Example 2. *For the same setting with Example 1, when* m *is set as* 2*, the string tokens will be produced as*

$$\boldsymbol{s} = \{\text{“pos1val0.12”}, \text{“pos2val-0.24”}\}$$

with only the first and second entries being selected; and when m is set as 1, the string tokens will be produced as

$$\boldsymbol{s} = \{\text{“}pos2val\text{-}0.24\text{”}\},$$

with only the second entry being selected.

Drawbacks. Although the filtering strategy is suggested to maintain a good balance between feature sparsity and search quality [46, 47], it might not be the best practice to reduce the number of string tokens with respect to finding nearest neighbors in general. First, for two points $\hat{\boldsymbol{x}}, \boldsymbol{x} \in \mathbb{R}^d$, their Euclidean distance

$$\|\hat{\boldsymbol{x}} - \boldsymbol{x}\|_2^2 = \sum_{i=1}^{d} (\hat{x}_i - x_i)^2, \tag{3.1}$$

is summed along each axis equally rather than biasedly based on the magnitude of $\hat{x}_i$ (or x_i). In specific, a mismatch/match with a (rounded) value 0.01 does not imply that it is less important than a mismatch/match with a 0.99, in terms of their contributions to the sum (3.1). What essentially matters is the deviation $\Delta_i := \hat{x}_i - x_i$ rather than the value of $\hat{x}_i$ (or x_i) by itself. Therefore, entries with small magnitude should not be considered as less essential and be totally ignored. Second, the efficacy of the filtering strategy is vulnerable to data distributions. For example, when the embedding vectors are binary codes [24, 31, 34, 35, 52], choosing top-m entries will lead to an immediate tanglement.

In the next subsection, we will propose an alternative encoder, which keeps all value information into consideration and is also more robust with respect to the underlying data distribution.

3.2 Subvector-wise Clustering

Different from the element-wise rounding one, an encoder that operates on a subvector level will be presented in this part. The idea is also quite natural and straightforward. For any vector $\boldsymbol{x} \in \mathbb{R}^d$, we divide it into m subvectors[1],

$$[\underbrace{x_1, \ldots, x_{d/m}}_{\boldsymbol{x}^1}, \underbrace{x_{d/m+1}, \ldots, x_{2d/m}}_{\boldsymbol{x}^2}, \ldots\ldots, \underbrace{x_{d-m+1}, \ldots, x_m}_{\boldsymbol{x}^m}]. \tag{3.2}$$

Denote $\mathcal{X}^i := \{\boldsymbol{x}_1^i, \boldsymbol{x}_2^i, \ldots, \boldsymbol{x}_n^i\}$ as the collection of the i-th subvectors from $\mathcal{X}$ for $i = 1, 2, \ldots, m$. We will then separately apply the classical k-means algorithm [37] to divide each $\mathcal{X}^i$ into k clusters with the learned assignment function

$$\mathcal{A}^i : \mathbb{R}^{d/m} \to \{1, 2, \ldots, k\}$$

assigning each subvector to the cluster index it belongs to. Then for any $\boldsymbol{x} \in \mathbb{R}^d$, we will encode it into a collection of m string tokens

$$\{\text{“pos1cluster}\{\mathcal{A}^1(\boldsymbol{x}^1)\}\text{”}, \text{“pos2cluster}\{\mathcal{A}^2(\boldsymbol{x}^2)\}\text{”}, \ldots\}. \tag{3.3}$$

The whole idea is illustrated in Figure 3. The trade-off between search latency and quality is well controlled by the parameter m. In specific, a larger m will tend to increase the search quality as well as the search latency, as more string tokens per each vector will be indexed.

[1]For simplicity, we assume m divides d.

In contrast with the element-wise rounding encoder, our subvector-wise clustering encoder obtains m string tokens without throwing away any entry in $\boldsymbol{x}$, and will generate string tokens more adaptive with the data distribution, as the assignment function $\mathcal{A}^i(\cdot)$ for each subspace is learned through $\mathcal{X}^i$ (or data points sampled from $\mathcal{X}^i$).

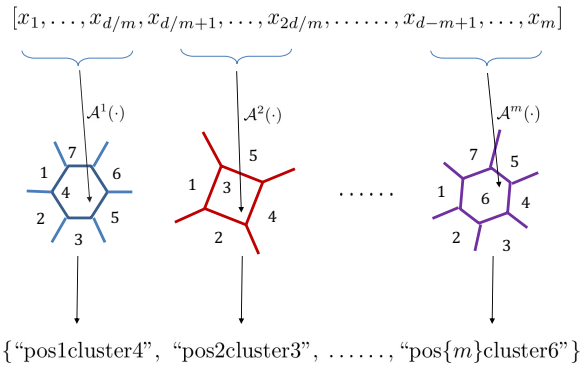

Figure 3: Illustration of the subvector-wise clustering encoder. The vector $\boldsymbol{x} \in \mathbb{R}^d$ is divided into m subvectors. Subvectors at the same position are considered together to be classified into k clusters. Then each subvector is encoded into a string token by combining its position in $\boldsymbol{x}$ and the cluster it belongs to, so exacly m string tokens will be produced.

4 EXPERIMENT

In this section, we will compare the performance of the subvector-wise clustering encoder and the element-wise rounding one in terms of both precision and latency, when they are being used in our content-based image retrieval system built upon Elasticsearch.

Settings. Our image datasets consists of around half a million images selected from Jet.com's furniture catalog [28]. For each image, we extract its image feature vector using the pre-trained Inception-ResNet-V2 model [53]. In specific, each image is embedded into a vector in $\mathbb{R}^{1536}$ by taking the output from the penultimate layer (i.e., the last average pooling layer) of the neural network model. String tokens are produced respectively with encoding schemes at different configurations. For the element-wise rounding encoder, we select $p \in \{0, 1, 2, 3\}$, and $m \in \{32, 64, 128, 256\}$. For the subvector-wise clustering encoder, we experiment with $k \in \{32, 64, 128, 256\}$ and $m \in \{32, 64, 128, 256\}$. Under each scenario, we index the image feature vectors and their string tokens into a single-node Elustersearch cluster deployed on a Microsoft Azure virtual machine [40] with 12 cores and 112 GiB of RAM. To better focus on the comparison of the efficacy in encoding scheme, only vanilla setting of Elasticsearch (one shard and zero replica) is used in creating each index.

Evaluation. To evaluate the two encoding schemes, we randomly select 1,000 images to act as our visual queries. For each of the query image, we find the set of its 24 nearest

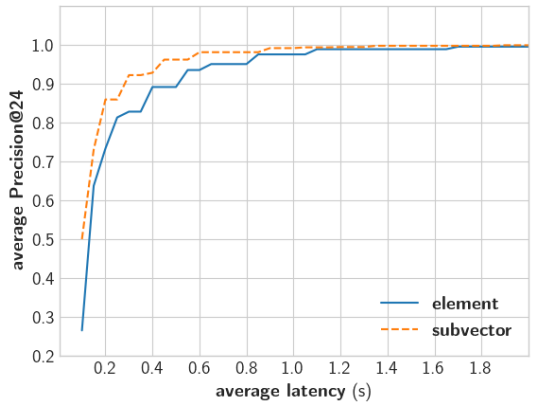

Figure 4: Pareto frontier for the element-wise rounding and the subvector-wise clustering encoders in the space of latency and precision. It can be clearly seen that our subvector-wise encoding scheme is capable of achieving higher precision with smaller latency.

neighbors in Euclidean distance, which is treated as gold standard. We use Precision@24 [49], which measures the overlap between the 24 images retrieved from Elasticsearch (with $r \in \{24, 48, 96, \ldots, 6144\}$ respectively) and the gold standard, to evaluate the retrieval efficacy of different encoding methods under various settings. We also record the latency for Elasticsearch to execute the retrieval and reranking steps in the searching phase.

Results. In Table 1, we report the Precision@24 and search latency averaged over the 1,000 queries randomly selected. Results corresponding to $p \in \{2, 3\}$ or $r \in \{24, 48\}$ are skipped as they are largely outperformed by other settings. Configurations that can achieve precision $\geq 80\%$ and latency ≤ 0.5s are highlighted in bold. From Table 1, we can see that the subvector-wise encoder outperforms the element-wise one, as for all results obtained by the element-wise encoder, we can find a better result from the subvector-wise one in both precision and latency. To better visualize this fact, we plot the *Pareto frontier curve* over the space of precision and latency in Figure 4. In specific, the dashed (resp. solid) curve in Figure 4 is plotted as the best average Precision@24 achieved among all configurations we experiment for element-wise rounding (resp. subvector-wise clustering) encoder, under different latency constraints. From Figure 4, we can more clearly observe that the subvector-wise encoder surpasses the element-wise one. Notably, when we require the search latency to be smaller than 0.3 second, the subvector-wise encoder is able to achieve an average Precision@24 as 92.14%, yielding an improvement of more than 11% over the best average Precision@24 that can be obtained by the element-wise one.

5 FUTURE WORK

Although our subvector-wise clustering encoder outperforms the element-wise rounding one, it might be still restrictive to enforce a vector to be divided into subvectors exclusively using (3.2), which could potentially downgrade the performance of the encoder. Our next step is to preprocess the data (e.g., transform the data through some linear operation $\boldsymbol{x} \mapsto \mathcal{T}[\boldsymbol{x}]$ with $\mathcal{T}[\cdot]$ learned from the data) before applying our subvector-wise clustering encoder. We believe this flexibility will make our encoding scheme more robust and adaptive with respect to different image feature vectors extracted from various image descriptors. Another interesting research direction is to evaluate the performances of different encoding schemes in other information retrieval contexts–e.g., *neural ranking model based textual searches* [5, 41, 50], where relevances between user-issued queries and catalog products are modeled by their Euclidean distances in the embedding space to better match customers' intents with products.

ACKNOWLEDGEMENT

We are grateful to three anonymous reviewers for their helpful suggestions and comments that substantially improve the paper. We would also like to thank Eliot P. Brenner and Aliasgar Kutiyanawala for proofreading the first draft of the paper.

REFERENCES

[1] A. Aksyonoff. 2011. *Introduction to Search with Sphinx: From installation to relevance tuning.* " O'Reilly Media, Inc.".
[2] A. Andoni and P. Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of FOCS*. 459–468.
[3] Y. Bengio. 2009. Learning deep architectures for AI. *Foundations and trends® in Machine Learning* 2, 1 (2009), 1–127.
[4] A. Bialecki, R. Muir, G. Ingersoll, and L. Imagination. 2012. Apache lucene 4. In *Proceedings of SIGIR workshop on open source information retrieval*.
[5] E. P. Brenner, J. Zhao, A. Kutiyanawala, and Z. Yan. 2018. End-to-End Neural Ranking for eCommerce Product Search. In *Proceedings of SIGIR eCom'18*.
[6] Elasticsearch contributors. 2016. Multi-dimensional points, coming in Apache Lucene 6.0. Retrieved June 16, 2018 from https://www.elastic.co/blog/lucene-points-6.0.
[7] Elasticsearch contributors. 2017. Numeric and Date Ranges in Elasticsearch: Just Another Brick in the Wall. Retrieved June 16, 2018 from https://www.elastic.co/blog/numeric-and-date-ranges-in-elasticsearch-just-another-brick-in-the-wall.
[8] Elasticsearch contributors. 2018. Full text queries. Retrieved May 01, 2018 from https://www.elastic.co/guide/en/elasticsearch/reference/6.1/full-text-queries.html.
[9] Elasticsearch contributors. 2018. Function score query. Retrieved May 01, 2018 from https://www.elastic.co/guide/en/elasticsearch/reference/6.1/query-dsl-function-score-query.html.
[10] Elasticsearch contributors. 2018. Plugins. Retrieved May 01, 2018 from https://www.elastic.co/guide/en/elasticsearch/reference/6.1/modules-plugins.html.
[11] Elasticsearch contributors. 2018. Range query. Retrieved May 01, 2018 from https://www.elastic.co/guide/en/elasticsearch/reference/6.1/query-dsl-range-query.html.
[12] Elasticsearch contributors. 2018. Rescoring. Retrieved May 01, 2018 from https://www.elastic.co/guide/en/elasticsearch/reference/6.1/search-request-rescore.html.
[13] Elasticsearch contributors. 2018. Stories from Users Like You. Retrieved May 01, 2018 from https://www.elastic.co/use-cases.
[14] Elasticsearch contributors. 2018. Term query. Retrieved May 01, 2018 from https://www.elastic.co/guide/en/elasticsearch/reference/6.1/query-dsl-term-query.html.
[15] Lucene contributors. 2018. Apache Lucene library. Retrieved May 06, 2018 from https://lucene.apache.org.
[16] R. Datta, D. Joshi, J. Li, and J. Wang. 2008. Image retrieval: Ideas, influences, and trends of the new age. *Comput. Surveys* 40, 2 (2008), 5.
[17] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. 2014. Decaf: A deep convolutional activation feature for generic visual recognition. In *Proceedings of ICML*. 647–655.

r	Encoding	Round./Cluster.	# of feature tokens (m)			
			32	64	128	256
96	element	0-decimal-place	53.43% | 0.1237	64.35% | 0.2339	76.44% | 0.5256	88.64% | 1.5342
	element	1-decimal-place	26.56% | 0.0920	37.94% | 0.1592	50.35% | 0.3370	63.71% | 0.8207
	subvector	32-centroids	34.80% | 0.1111	54.45% | 0.1914	74.22% | 0.3760	87.44% | 0.8914
	subvector	64-centroids	39.52% | 0.0963	58.51% | 0.1630	76.70% | 0.3426	87.28% | 0.7563
	subvector	128-centroids	44.43% | 0.0914	61.93% | 0.1544	78.89% | 0.3088	85.58% | 0.7186
	subvector	256-centroids	50.00% | 0.0900	66.22% | 0.1480	79.05% | 0.2970	82.89% | 0.6757
192	element	0-decimal-place	63.72% | 0.1405	74.63% | 0.2499	85.38% | 0.5416	94.13% | 1.5536
	element	1-decimal-place	32.49% | 0.1084	45.50% | 0.1748	59.05% | 0.3529	72.12% | 0.8424
	subvector	32-centroids	43.73% | 0.1256	64.88% | 0.2080	**83.13% | 0.3917**	93.56% | 0.9146
	subvector	64-centroids	48.84% | 0.1130	69.14% | 0.1795	**85.12% | 0.3594**	93.28% | 0.7745
	subvector	128-centroids	55.14% | 0.1082	72.62% | 0.1714	**87.08% | 0.3250**	91.97% | 0.7367
	subvector	256-centroids	61.41% | 0.1066	77.08% | 0.1644	**87.32% | 0.3137**	89.28% | 0.6915
384	element	0-decimal-place	73.30% | 0.1749	**82.76% | 0.2852**	91.19% | 0.5756	97.03% | 1.5963
	element	1-decimal-place	38.94% | 0.1431	53.43% | 0.2093	67.12% | 0.3877	79.25% | 0.8741
	subvector	32-centroids	53.37% | 0.1603	73.92% | 0.2417	**89.06% | 0.4262**	96.82% | 0.9509
	subvector	64-centroids	59.01% | 0.1479	78.15% | 0.2139	**91.25% | 0.3935**	96.59% | 0.8097
	subvector	128-centroids	66.20% | 0.1433	**81.56% | 0.2061**	**92.75% | 0.3596**	95.44% | 0.7705
	subvector	256-centroids	73.01% | 0.1415	**85.88% | 0.1995**	**92.67% | 0.3520**	93.38% | 0.7243
768	element	0-decimal-place	**81.27% | 0.2455**	**89.09% | 0.3547**	94.98% | 0.6443	98.60% | 1.6613
	element	1-decimal-place	45.83% | 0.2130	61.30% | 0.2801	74.60% | 0.4574	84.87% | 0.9427
	subvector	32-centroids	63.45% | 0.2297	**81.30% | 0.3117**	**93.40% | 0.4974**	98.58% | 1.0195
	subvector	64-centroids	69.01% | 0.2182	**85.47% | 0.2837**	**95.41% | 0.4647**	98.38% | 0.8798
	subvector	128-centroids	76.70% | 0.2133	**88.91% | 0.2762**	**96.13% | 0.4288**	97.50% | 0.8402
	subvector	256-centroids	**83.55% | 0.2112**	**92.14% | 0.2701**	**95.90% | 0.4267**	95.94% | 0.7970
1536	element	0-decimal-place	**87.55% | 0.3923**	93.45% | 0.5027	97.47% | 0.8012	99.29% | 1.8486
	element	1-decimal-place	53.76% | 0.3656	68.68% | 0.4361	81.05% | 0.6069	89.48% | 1.0931
	subvector	32-centroids	72.75% | 0.3703	**87.30% | 0.4524**	96.14% | 0.6400	99.36% | 1.1574
	subvector	64-centroids	78.85% | 0.3581	**91.52% | 0.4218**	97.74% | 0.6045	99.28% | 1.0188
	subvector	128-centroids	**86.00% | 0.3537**	**94.12% | 0.4158**	98.03% | 0.5665	98.60% | 0.9763
	subvector	256-centroids	**91.16% | 0.3512**	**95.97% | 0.4087**	97.70% | 0.5582	97.44% | 0.9281
3072	element	0-decimal-place	92.38% | 0.6843	96.40% | 0.8166	98.80% | 1.0909	99.63% | 2.1638
	element	1-decimal-place	61.50% | 0.6625	75.62% | 0.7380	86.32% | 0.9135	92.85% | 1.3946
	subvector	32-centroids	81.25% | 0.6645	92.11% | 0.7483	97.95% | 0.9375	99.68% | 1.4589
	subvector	64-centroids	87.82% | 0.6556	96.32% | 0.7131	99.00% | 0.9006	99.68% | 1.3189
	subvector	128-centroids	93.26% | 0.6508	97.72% | 0.7126	99.08% | 0.8604	99.21% | 1.2756
	subvector	256-centroids	96.06% | 0.6470	97.94% | 0.7074	98.72% | 0.8566	98.37% | 1.2230
6144	element	0-decimal-place	95.52% | 1.2630	98.22% | 1.3778	99.45% | 1.6737	99.82% | 2.7669
	element	1-decimal-place	68.26% | 1.2535	81.75% | 1.2942	90.69% | 1.4800	95.24% | 1.9542
	subvector	32-centroids	89.61% | 1.2081	95.86% | 1.2938	99.10% | 1.4892	99.85% | 2.0124
	subvector	64-centroids	95.43% | 1.2031	98.87% | 1.2537	99.65% | 1.4459	99.82% | 1.8647
	subvector	128-centroids	97.56% | 1.1985	99.13% | 1.2565	99.54% | 1.3959	99.52% | 1.8200
	subvector	256-centroids	98.20% | 1.1957	98.90% | 1.2542	99.25% | 1.4037	98.97% | 1.7586

Table 1: Mean Precision@24 | ES average latency. For each setting, we average the Precision@24 and the number of seconds used over the 1,000 query images randomly selected from the furniture dataset. Settings with mean precision $\geq 80\%$ and latency $\leq 0.5s$ are highlighted in bold.

[18] T. Ge, K. He, Q. Ke, and J. Sun. 2013. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of CVPR*. 2946–2953.

[19] C. Gennaro, G. Amato, P. Bolettieri, and P. Savino. 2010. An approach to content-based image retrieval based on the Lucene search engine library. In *Proceedings of TPDL*. 55–66.

[20] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. 2013. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 12 (2013), 2916–2929.

[21] I. Goodfellow, Y. Bengio, and A. Courville. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.

[22] C. Gormley and Z. Tong. 2015. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine.* " O'Reilly Media, Inc.".

[23] K. He, F. Wen, and J. Sun. 2013. K-means hashing: An affinity-preserving quantization method for learning binary compact codes. In *Proceedings of CVPR*. 2938–2945.

[24] J. Heinly, E. Dunn, and J. Frahm. 2012. Comparative evaluation of binary features. In *ECCV*. 759–773.

[25] H. Jegou, M. Douze, and C. Schmid. 2011. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2011), 117–128.

[26] H. Jegou, F. Perronnin, M. Douze, J. Sánchez, P. Perez, and C. Schmid. 2012. Aggregating local image descriptors into compact codes. *IEEE transactions on pattern analysis and machine intelligence* 34, 9 (2012), 1704–1716.

[27] H. Jegou, R. Tavenard, M. Douze, and L. Amsaleg. 2011. Searching in one billion vectors: re-rank with source coding. In *Proceedings of ICASSP*. IEEE, 861–864.

[28] Jet.com. 2018. Furniture. Retrieved May 01, 2018 from https://jet.com/search?category=18000000.

[29] Y. Kalantidis and Y. Avrithis. 2014. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of CVPR*. 2321–2328.

[30] Nicholas Knize. 2018. Geo Capabilities in Elasticsearch. Retrieved June 16, 2018 from https://www.elastic.co/assets/blt827a0a9db0f2e04e/webinar-geo-capabilities.pdf.

[31] H. Lai, Y. Pan, Y. Liu, and S. Yan. 2015. Simultaneous feature learning and hash coding with deep neural networks. In *Proceedings of CVPR*. 3270–3278.

[32] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng. 2012. Building high-level features using large scale unsupervised learning. In *Proceedings of ICML*. 507–514.

[33] W. Liu, C. Mu, S. Kumar, and S. Chang. 2014. Discrete graph hashing. In *Advances in NIPS*. 3419–3427.

[34] M. Loncaric, B. Liu, and R. Weber. 2018. Convolutional Hashing for Automated Scene Matching. *arXiv preprint arXiv:1802.03101* (2018).

[35] X. Lu, L. Song, R. Xie, X. Yang, and W. Zhang. 2017. Deep Binary Representation for Efficient Image Retrieval. *Advances in Multimedia* 2017 (2017).

[36] M. Lux, M. Riegler, P. Halvorsen, K. Pogorelov, and N. Anagnostopoulos. 2016. LIRE: open source visual information retrieval. In *Proceedings of MMSys*.

[37] J. MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA, 281–297.

[38] M. McCandless, E. Hatcher, and O. Gospodnetic. 2010. *Lucene in action: covers Apache Lucene 3.0.* Manning Publications Co.

[39] G. Mesnil, Y. Dauphin, X. Glorot, S. Rifai, Y. Bengio, I. Goodfellow, E. Lavoie, X. Muller, G. Desjardins, D. Warde-Farley, P. Vincent, A. Courville, and J. Bergstra. 2011. Unsupervised and transfer learning challenge: a deep learning approach. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*. JMLR. org, 97–111.

[40] Microsoft Azure. 2018. Virtual machines. Retrieved May 01, 2018 from https://azure.microsoft.com/en-us/services/virtual-machines/.

[41] B. Mitra and N. Craswell. 2017. Neural Models for Information Retrieval. *arXiv preprint arXiv:1705.01509* (2017).

[42] M. Oquab, L. Bottou, I. Laptev, and J. Sivic. 2014. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of CVPR*. 1717–1724.

[43] R. Raina, A. Battle, H. Lee, B. Packer, and A. Ng. 2007. Self-taught learning: transfer learning from unlabeled data. In *Proceedings of ICML*. 759–766.

[44] A. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. 2014. CNN features off-the-shelf: an astounding baseline for recognition. In *CVPR workshop*. 512–519.

[45] S. Robertson and H. Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[46] M. Ruzicka, V. Novotny, P. Sojka, J. Pomikalek, and R. Rehurek. 2018. Flexible Similarity Search of Semantic Vectors Using Fulltext Search Engines. *http://ceur-ws.org/Vol-1923/article-01.pdf* (2018).

[47] J. Rygl, J. Pomikalek, R. Rehurek, M. Ruzicka, V. Novotny, and P. Sojka. 2017. Semantic Vector Encoding and Similarity Search Using Fulltext Search Engines. In *Proceedings of the 2nd Workshop on Representation Learning for NLP*. 81–90.

[48] G. Salton, A. Wong, and C. Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[49] H. Schütze, C. D. Manning, and P. Raghavan. 2008. *Introduction to information retrieval.* Vol. 39. Cambridge University Press.

[50] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil. 2014. A latent semantic model with convolutional-pooling structure for information retrieval. In *Proceedings of CIKM*. 101–110.

[51] D. Smiley, E. Pugh, K. Parisa, and M. Mitchell. 2015. *Apache Solr enterprise search server.* Packt Publishing Ltd.

[52] J. Song. 2017. Binary Generative Adversarial Networks for Image Retrieval. *arXiv preprint arXiv:1708.04150* (2017).

[53] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of AAAI*. 4278–4284.

[54] A. Torralba, R. Fergus, and Y. Weiss. 2008. Small codes and large image databases for recognition. In *Proceedings of CVPR*. 1–8.

[55] J. Wang, S. Kumar, and S. Chang. 2010. Semi-supervised hashing for scalable image retrieval. In *Proceedings of CVPR*. 3424–3431.

[56] Y. Weiss, A. Torralba, and R. Fergus. 2009. Spectral hashing. In *Advances in NIPS*. 1753–1760.

[57] Wikipedia contributors. 2018. Elasticsearch. Retrieved May 06, 2018 from https://en.wikipedia.org/wiki/Elasticsearch.

[58] Wikipedia contributors. 2018. Tf-idf. Retrieved May 06, 2018 from https://en.wikipedia.org/wiki/tf-idf.

[59] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. 2014. How transferable are features in deep neural networks. In *Advances in NIPS*. 3320–3328.

[60] M. Zeiler and R. Fergus. 2014. Visualizing and understanding convolutional networks. In *Proceedings of ECCV*. 818–833.